HardCilk: Cilk-like Task Parallelism for FPGAs

Mohamed Shahawy, Canberk Sönmez, Cemalettin Belentepe, and Paolo Ienne
Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences, Lausanne, Switzerland
{first}.{last}@epfl.ch

***Abstract*—*High-level synthesis* (HLS) helps to develop hardware accelerators for *field-programmable gate arrays* (FPGAs) using C/C++ descriptions. HLS is tailored to exploit instruction-level parallelism and, where available, data-level parallelism in applications. Yet, some applications mostly display another type of parallelism known as *task-level parallelism* (TLP): they may have massive amounts of available parallelism but parallelizable execution threads follow starkly different control paths or make completely independent memory accesses. Alas, there is very limited support for TLP in HLS tools (with much of the support being for statically scheduled coarse-grained tasks) whereas TLP is widely supported on conventional CPU platforms via libraries like OpenCilk, Intel Threading Building Blocks, and OpenMP. In this paper, we introduce a framework for supporting software-like TLP on FPGAs. The framework provides a parameterized architectural template that implements all hardware modules needed to support TLP primitives and task management. The emphasis is on providing programmers with a software-like experience: limited hardware resource constraints should only impact performance and never functionality. For this, all queues in the limited BRAMs are virtually extended to larger HBM/DDR FPGA memory and potentially beyond. We provide an open-source Chisel hardware generator that creates a dedicated task management system from a given Cilk-like application description. It is then straightforward to integrate it with HLS-based processing elements to realize full TLP-enabled applications on FPGAs. In the evaluation, we focus on the efficiency and scalability of our hardware task management system and compare it with OpenCilk, a recent software TLP framework. The results show that the architectural building blocks consume a small percentage of resources on modern FPGAs designed for data centres and achieve linear hardware scalability. For a reasonable range of parameters, the system shows near-perfect efficiency and speedup scales linearly when increasing the number of processing elements.**

I. Introduction

Parallelism is key for software programmers to extract performance from their code. Traditional high-end CPUs automatically exploit *instruction-level parallelism* (ILP) in hardware. *High-level synthesis* (HLS) tools do something very similar when producing circuits from software code.

To go beyond ILP, programmers need to express opportunities for parallel execution through languages or libraries. Arguably, the most common form of parallelism is data-level parallelism, where programmers express that the exact same series of computations is repeated over a large set of data. It leads to parallel execution through vector and multimedia instruction sets of classic CPUs, but is the main target of an immensely popular class of devices: GPUs. Their ubiquity is a combination of the importance of data-level parallelism in applications and of the availability of appropriate languages like CUDA. Data-level parallelism is well supported also by HLS tools for FPGAs, but it is not always self-evident that, in this context, FPGAs can have a performance edge over extremely optimized GPUs.

There is a completely other type of parallelism that is well supported through programming languages and libraries: *task-level parallelism* (TLP). It is rather poorly supported in FPGA tools (especially for fine grain tasks) and our goal is to change this, for we believe that it has the potential to make FPGAs stand out from other powerful computing devices.

A. Why We Need TLP

The whole idea of TLP is to decompose computation in individual independent tasks and execute them on the largest possible number of *processing elements* (PE). If these tasks followed the very same execution path (and in many practical applications they are guaranteed to do so), data-level parallelism would be the paradigm of choice. Yet, there are many important and massively parallel applications where individual tasks can be expressed through the same code (often small and recursive functions) but each individual execution is completely different (*control dominated*) or accesses totally distinct sections of large data structures.

A classic example is applications on very large graphs: Many performance-optimal algorithms perform the same recursive set of operations starting from all nodes of the graph (e.g., some local search mostly confined to the neighbourhood of the starting point). While, for very large graphs with millions or billions of nodes, there is a gigantic amount of exploitable parallelism, executions from different starting nodes have, in general, absolutely nothing in common; these algorithms are naturally a terrible fit for GPUs. With the recent attention to efficient graph analytics for ML/AI applications, some authors have recognized that asymptotically optimal algorithms for a variety of important graph explorations have a practical edge, perhaps predictably, over suboptimal algorithms that happen to be easily portable to GPUs [1], [2]. The key to the success of these algorithms is the pervasiveness of large multicore CPUs and the availability of powerful TLP libraries like OpenCilk, Intel Threading Building Blocks, and OpenMP. Nothing truly similar is available for FPGAs.

B. TLP Is a Perfect Fit for FPGAs

We think that fine-grain TLP is an immensely promising paradigm to open FPGAs to software programmers: (1) Firstly, as mentioned, it is easily accessible as a programming paradigm and applications can be tested in software with an extremely fast development turnaround, especially when

2576-2621/24/$31.00 ©2024 IEEE
DOI 10.1109/FCCM60383.2024.00025

compared to FPGAs. (2) Secondly, tasks are often a very limited set of small functions called recursively millions or billions of times. It is likely that fairly small iterative functions can be implemented efficiently out-of-the-box by HLS tools. Even if not, at production time, hardware engineers can optimize at *register-transfer level* (RTL) these small functions—if there are only few, if they are small, and if the rest of the system is unmodified, it is a minor nonrecurring engineering cost. (3) Finally, FPGAs seem to be an ideal match to TLP applications. While CPUs can also be very successful, the implementation in hardware of small kernel functions gives a potentially great advantage to FPGAs over the cores of CPUs. But, perhaps even more importantly, TLP applications on CPUs spend a considerable amount of time and resources (through the functionality implemented in OpenCilk, Intel Threading Building Blocks, or OpenMP) to manage, schedule, and load-balance tasks of immensely variable execution time. It has been shown that performance-competitive uses of TLP on multicore CPUs, besides a fairly intuitive rewriting of the original sequential algorithms in TLP form, need fairly heavy restructuring not to incur catastrophic task management overheads [1], [2]. On FPGAs, the whole management of tasks and of their returned results could also be efficiently implemented in dedicated circuitry, thus not subtracting any valuable compute cycle from the PEs. The possibility of achieving this generically and economically for any TLP application is the purpose of this work.

C. Our Goal with HardCilk

Surprisingly little has been done to support generic forms of fine-grain TLP on FPGAs. One of the few examples and perhaps the closest to our aims was the pioneering work on ParallelXL [3]. While we see that work as inspiring, we also view it with a couple of important limitations: (a) On the one hand, the architecture it implements is very much borrowed from traditional multicores and misses key opportunities to develop concurrent hardware structures for the tasks of a particular TLP application. (b) On the other hand, it has a defect typical of many hardware contraptions on FPGAs; namely, some resources are naturally limited (e.g., queue sizes) and applications that consume them completely either fail to run or result in incorrect execution. In this paper, we introduce HardCilk: an open-source Chisel generator [4] of RTL designs that, given information that a compiler could easily extract from software TLP code (e.g., written in OpenCilk), produces a complete dedicated task-management system for FPGAs. Our philosophy is to provide the programmer with a software-like experience: limits in available resources must only result in (possibly minimal) performance losses and never in failure to compile or, worse, to produce the correct result. Our aim is to show that it is possible with moderate resources to manage millions of tasks efficiently, often achieving quasi-optimal utilization of the processing units.

The rest of the paper is organized as follows: The next section introduces the computational model that we will target with HardCilk. Section III describes the template architecture of our system. We evaluate the efficiency and scalability of HardCilk in Section IV. The paper concludes after a review of the state of the art in Section V.

II. Computational Model

This section describes the programming model used by our framework. Cilk [5] is one of the very first TLP frameworks developed for multicore CPUs, introducing a provably efficient task scheduler with a nonblocking programming model. The Cilk framework was implemented as a C-language extension. It later developed into Cilk-5 [6] with a redesigned language and new engineering of the underneath functionality. Other versions appeared, such as Cilk Plus by Intel and most recently OpenCilk [7]. These later reincarnations of Cilk still implement the main Cilk principles; however, they use more sophisticated runtime systems to implement a more friendly programming model. As a result, as shown by Joerg [8], one can translate a program written in the original Cilk syntax to the syntax of the more friendly Cilk-5 programming interface. In this paper we focus on the classic Cilk programming model which is particularly suitable for hardware implementation for its nonblocking execution model, leaving the actual integration in a modern compiler for future work. Our focus here is to prove that task management requires only a handful of fairly simple and composable hardware building blocks. The composition of these blocks into an efficient network for a given application can be obtained directly from Cilk code.

A. Task Representation

The first component of the programming model is the task representation. A *task* is a set of arguments *args* that represent the input to a function f with a continuation k. A *continuation* is a pointer to another task that this task shall return to; this will be further explained in the next section. Cilk tasks are either ready or pending. Tasks are *ready* when they have all their arguments available. When a task is created with a number of missing arguments, it is *pending* and initialized with a join counter j equal to the number of missing arguments. A task is only ready and scheduled when all the arguments are available and the join counter is equal to zero. We use explicit continuation passing style to implement nonblocking HLS-friendly joins for FPGA TLP accelerators.

B. Expressing Parallelism

Cilk provides the *spawn* keyword as a tool for a task to generate a child task that could be executed in parallel. This is similar to a function call; however, the spawned child could execute in parallel to the parent node. Spawning does not mandate parallel execution; it only indicates to the scheduler that, if there are workers available, it is legal to execute that task in parallel. In a fork-join parallelism representation, *forks* are represented by spawning in the Cilk model.

Normally, with software fork-join parallelism, *joins* are implemented as blocking statements where the operating system could suspend the waiting thread and schedule the threads that will complete the join. Since we are interested in implementing

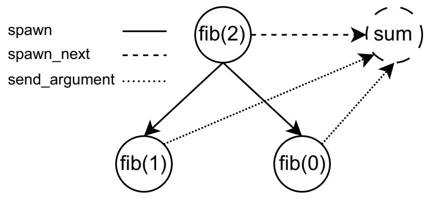

Fig. 1: An example execution graph for the Fibonacci example with $n = 2$.

this model in hardware, we observe that suspending the execution of an arbitrary PE is a nontrivial design task and is not supported by any HLS tool we know. Therefore, we use explicit continuation passing to implement nonblocking joins: *Explicit continuation passing* is a style of programming used in functional programming where control is explicitly passed in the form of continuations; if a task has a join, the programmer has to break the task into two tasks. The first task executes the instructions before the join, and the second executes the instructions after the join. In addition to executing the instructions before the join, the first task informs the scheduler that an instance of the second task should be executed in the future. For that, the scheduler allocates an empty placeholder for the second task, known as a successor task, in memory and returns a continuation—i.e., an address referring to that placeholder—that some task needs to return to at some point during the execution. The first task can now create its children, using spawn, passing them the allocated continuation where they need to return. Finally, the first task can now return without waiting for its children and the children are the ones responsible for returning to the successor task.

Cilk provides the *spawn_next* and *send_argument* keywords to create successors and return to them. The spawn_next primitive informs the runtime system that this task should be initialized with a specific join counter and scheduled only when all the arguments are available. Children tasks use the send_argument primitive to return values to successor tasks and inform the runtime system to check the successor task join counter.

C. Fibonacci: A Classic Example

Calculating the n-th Fibonacci number recursively and in parallel using fork-join parallelism is a common choice to present the idea of TLP. Figure 2 shows the Cilk code implementing the algorithm. The fib function is divided into two tasks: the operations before the join and those after it. The join is represented using explicit continuation passing. The directed acyclic graph shown in Figure 1 shows the execution graph for the Fibonacci example with $n = 2$.

D. Work Stealing

Cilk uses a work-stealing scheduler [9]. Figure 3 shows the functionality of the scheduler. During execution, a PE might spawn multiple child tasks that could be executed independently. The spawned tasks are pushed to a local double-ended

```
task fib (cont int k, int n):
    if(n < 2):
        send_argument (k, n)
    else:
        cont int x, y
        spawn_next sum (k, ?x, ?y)
        spawn fib (x, n-1)
        spawn fib (y, n-2)

task sum (cont int k, int x, int y):
    send_argument (k, x+y)
```

Fig. 2: A Cilk program, consisting of two tasks, to compute the nth Fibonacci number [5]. When using a '?' symbol before a continuation variable, it indicates to the runtime that the function does not take the continuation itself as an argument but rather the value that will be returned to that continuation.

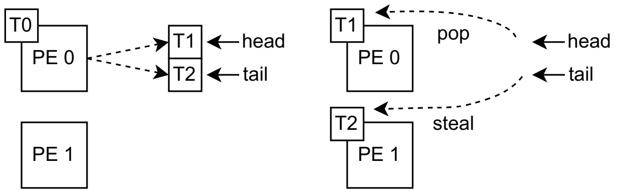

Fig. 3: Work stealing dynamically load-balances generated tasks during execution.

queue. The PE uses the head of its local queue to push and pop tasks. If other PEs do not have work to execute, they steal a task from the tail of the local queues of other PEs, known as victim PEs, that have ready tasks in their queues. This stealing is done randomly in software, but as we show in Sections III and IV, we exploit an approach more suitable for hardware.

III. Design

HardCilk follows two main design principles: Firstly, there should be no deadlocks due to hardware limitations. The virtualization of compute resources, specifically memory, has been exploited for a long time in traditional computing. We employ a similar paradigm by implementing extender modules that connect the system components to high-capacity FPGA memory resources, such as DDR and HBM, to extend the BRAM resources. Secondly, HardCilk supports an arbitrary number of PEs for each task type, for arbitrary Cilk applications with an arbitrary number of different tasks. We provide a configurable hardware generator [4] to automatically create the required architectural components and interconnect them. The user needs only to connect the PEs and the memory system to the generated system.

A. Architectural Template Overview

HardCilk uses an architectural template to implement the three keywords supported by the Cilk-like computational model. The architectural template consists of three building blocks: the work-stealing scheduler, the closure allocator, and the argument notifier. Together, they support the programming primitives of Cilk we target:

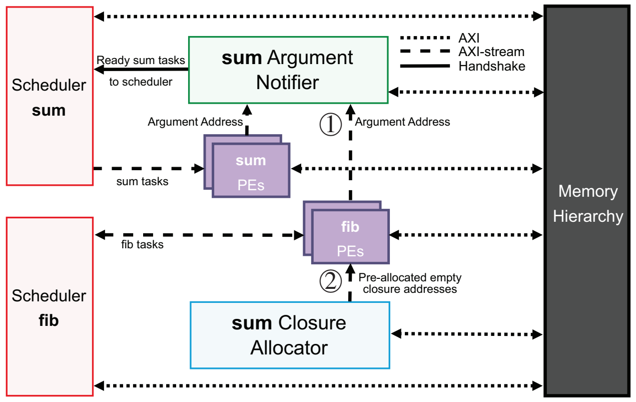

Fig. 4: HardCilk system for the Fibonacci code in Figure 2. HardCilk allows specializing PEs and the scheduler separately for each task type. Connections between components are dictated by send_argument and spawn_next relations between different types of tasks. For example, $fib \xrightarrow{\text{send\_argument}} sum$ corresponds to ①, and $fib \xrightarrow{\text{spawn\_next}} sum$ corresponds to ②.

- **Spawn.** The work-stealing scheduler is an architectural component that exposes two streaming interfaces to each PE. PEs can create new tasks by sending tasks to the scheduler. When ready to process a new task, they can obtain a ready task from the scheduler.
- **Spawn_next.** Cilk uses a data structure, known as *closure*, to express an empty placeholder of a successor task. Closures carry the arguments to the task and the join counter value that corresponds to the number of missing arguments (those that will be the object of a Send_argument). spawn_next requires the allocation of empty closures in main memory to enable creating successor tasks. A closure allocator provides each PE with preallocated addresses of empty closures for a specific type of task, so that PEs can issue a spawn_next.
- **Send_argument.** When a task generates an argument for a successor task, it simply writes the argument to memory in the closure. However, it needs to notify the system that an argument was written to decrement the join counter and verify whether the successor task has become ready. The argument notifier is an architectural element that provides this functionality and offers a streaming interface to each PE that may issue a send_argument, to be informed when this happens.

Figure 4 shows the corresponding HardCilk architectural template instance to the Fibonacci Cilk code in Figure 2.

These three architectural building blocks are designed using the same paradigm. Each has servers, clients, and a network: Servers implement the interface to main memory, typically to offload excess data. PEs connect to appropriate clients of the blocks corresponding to the needed operations. Finally, the network connects the servers and clients of each block.

The architectural generator creates the system as a collection of these architectural blocks, each replicated and customized

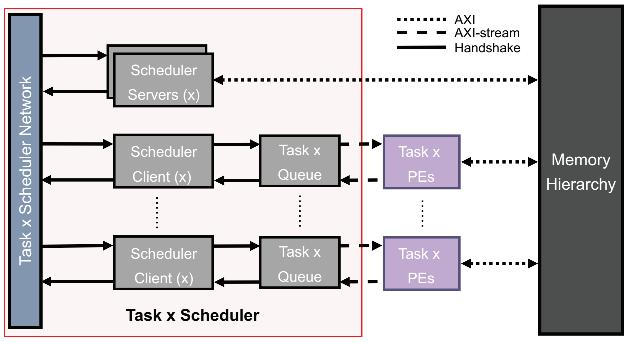

Fig. 5: Work stealing scheduler, generalized for an arbitrary task x. The scheduler clients manage the local task queues by (1) requesting tasks if the queues are empty and (2) offloading tasks to the network if they are almost full.

for the various tasks. Clients are replicated for all PEs requiring an interface and the networks sized accordingly. The overall connectivity depends on the relations between tasks within a Cilk code, such as which task can spawn which other task or which task sends an argument to which other task. Such static relations between task are straightforward to extract from Cilk code. Cilk keywords spawn and spawn_next have an identifier for the task type they issue. For instance, in line 7 in the code snippet for Fibonacci in Figure 2, the spawn statement specifies the type of task it creates; therefore, the relation $fib \xrightarrow{\text{spawn}} fib$ could be extracted. The send_argument primitive is a bit more ambiguous; however, the relation $fib \xrightarrow{\text{send\_argument}} sum$ could be extracted based on the unambiguous relation $fib \xrightarrow{\text{spawn\_next}} sum$.

B. Work Stealing Scheduler

Figure 5 shows the design of the work-stealing scheduler. The scheduler has a task queue and a scheduler client per PE, a number of scheduler servers, and a stealing network to connect clients and servers. The task queues are local BRAM-based double-ended queues that are directly connected to the PEs via streaming interfaces. The scheduler clients connect the task queues with the work-stealing network. The scheduler servers connect the network with memory-based task queues that could extend the hardware queues.

The interaction between the different scheduler clients serving different PEs implements a Cilk-inspired work-stealing policy. (i) If the queue is empty or near-empty, the client issues a task request over the work-stealing network and waits to receive a task. On receiving the task, the client pushes it to the queue to be executed by the corresponding PE. (ii) If the local queue has enough available tasks, the client listens to the network for any stealing requests. If it finds any, it pops the request from the network, pops a task from its local queue, and pushes it to the network. (iii) If the local queue of that client is near-full or full, the client pops a task from the queue and pushes it to the network without waiting for any requests.

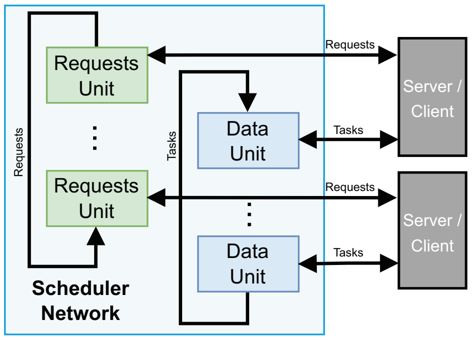

Fig. 6: Scheduler Network consists of two ring networks circulating task data and requests among servers and clients. Low resource utilization of ring networks helps system scalability.

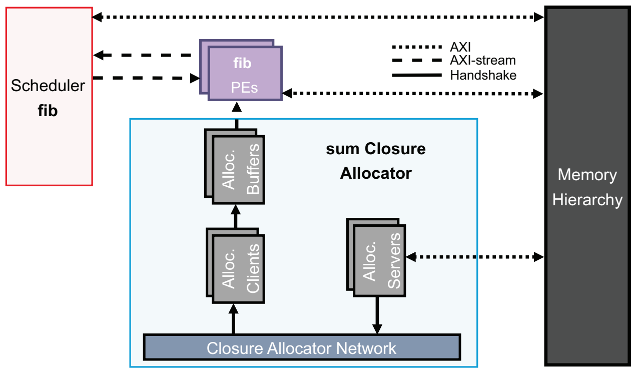

Fig. 7: Our system pre-allocates closures in bulk, rather than allocating them one-by-one. Closure Allocator delivers pre-allocated closure addresses from the memory to PEs.

This third functionality does not exist in software-based TLP, as they do not suffer from hardware queues of limited size.

The scheduler servers prevent the clients from saturating the network and from starving: (i) If they detect a contention in the network, they offload tasks to a memory-based task queue. (ii) If task requests are detected in the network, servers pop tasks from the memory queues and insert them to the network. Our architecture does not impose any restriction on the number of scheduler servers, and they can be increased to enhance the performance of the scheduler as they hide the latency of the memory accesses through parallel network provisioning. Such clients need initialization from the host with the base addresses of their private memory-based queues and the available size, which can be extended at runtime.

C. Scheduler Networks

The scheduler networks (Figure 6) are designed to be cheap and scalable: we have opted for two parallel ring networks communicating in opposite directions. Due to their topology, they are more efficient implementing a stealing policy different from the one used in software TLP. Doing random work-stealing in hardware would require point-to-point communication between the different stealing servers, adding much latency going through the ring network or requiring some form of a crossbar interconnect. We decided to use a different policy where if a stealing server wants a task, it issues a generic stealing request which is pushed to the first ring network, namely the requests network. The ring network takes this request and rotates it each cycle to a different client or server on the network. Some of these will have tasks in their local queues allowing them to serve these requests. A task is pushed on the other ring network (data) that flows in the opposite direction to the requests network; from there it would be popped by the requesting server. Each instance of the work stealing network connects to at least as many clients as there are PEs for the corresponding task. Yet, since PEs of a specific type are also allowed to spawn tasks of other types, the ring network is easily extended with clients for PEs of a different type, so that these can spawn (but not execute) tasks. All components are inexpensive, and the architecture is sufficiently scalable that it can be extended to hundreds of PEs.

D. Closure Allocator

The closure allocator, shown in Figure 7, allows PEs to issue the spawn_next operation described by the Cilk programming model. We divide the spawn_next operation into two parts. The first consists of providing base addresses of empty closures and is the task of the closure allocator. This distributes preallocated closure addresses of a specific task type to all the PEs that can spawn_next that task. The second part is to write the join counter and ready arguments values to the address of the empty closure. PEs should not block when writing to memory, so a write buffer handles the writes and frees immediately the PEs.

The closure allocator is inspired by the work-stealing scheduler, but significantly simpler. It too has a ring network, single in this case, where the allocation servers inject addresses of empty closures. All PEs sending arguments to a particular closure type have a closure client; this pops addresses from the ring and places them in a private buffer connected to the PE with a streaming interface. The purpose is that PEs, when executing a spawn_next operation, always have a ready closure address to insert into the task they spawn.

E. Arguments Notifier

The arguments notifier, shown in Figure 8, provides atomic access to the join counters of the closures waiting for arguments and schedules them when ready to execute. When a PE writes an argument to the closure of a waiting task, it must also announce this write to the argument notifier. The arguments notifier is yet another variations of the same scheme: a ring network, a number of servers, and clients for each PE that is sending an argument. The clients, one per PE executing a send_argument of a particular closure type, gather closure addresses from the PEs via streaming interfaces and inject them on the ring network. Servers pop closure addresses from the network and decrement and check the join counters of the closures. Modifying the counter must be implemented

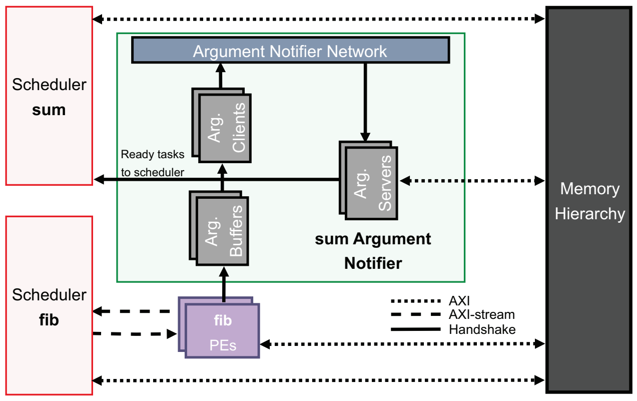

Fig. 8: Argument Notifier collects generated arguments from PEs to fill in the arguments of successor tasks. When all the arguments of a task are ready, the argument notifier moves it to the corresponding scheduler. We omit connections from sum PEs to the argument notifier for brevity.

TABLE I: Example configuration of HardCilk for Fibonacci.

fib	PEs = 16	Scheduler Servers = 4
sum	PEs = 8	Scheduler Servers = 4
	Closure Servers = 1	Argument Servers = 4
spawn	fib ⇒ fib	
spawn_next	fib ⇒ sum	
send_argument	fib ⇒ sum	sum ⇒ sum

atomically, so only one server can access a particular closure. We shard the addresses so that different servers can handle a subset of addresses atomically and in parallel. Finally, if a server determines that a task has become ready (i.e., its join counter is null), it injects the task into the appropriate work-stealing network. For this, the generator instantiates an additional scheduler client for each argument server.

F. HardCilk Generator

HardCilk is an open-source Chisel generator [4] that produces the complete shell RTL necessary for a Cilk application; the user has only the PEs, possibly designed by an HLS tool, and the memory interface. The generator needs a number of parameters to run, essentially from two classes: (a) Some express the relations between tasks that could be extracted automatically from Cilk code as mentioned in Section III-A and the nature of these tasks (size and nature in bits, etc.). (b) Other parameters influence the degree of parallelism and the performance of the system. These include the number of PEs of each type and the number of servers of each network. Table I shows an example configuration for the architectural generator that we have for the Fibonacci Cilk code.

IV. Evaluation

Evaluation methodology. In our evaluation, we use a number of synthetic benchmarks engineered to test individual parts of our system. Our benchmarks, while being synthetic, exhibit the parallelism characteristics of applications that we target—i.e., those with high degrees of control-dominated parallelism. We prefer focusing on system-level optimizations and characteristics rather than improving application-specific execution schemes, which are already explored by prior work [10], [3]. We use the definitions in Table II to characterize task parallel applications used in our evaluation.

TABLE II: Definitions of parameters used in the evaluation.

T_n	The actual measured runtime of the application on n processors
T'_n	The theoretical perfect speedup for a computation on n compute units ($= T_1/n$)
Efficiency (η)	The ratio between T'_n to T_n represents the efficiency of the scheduler to achieve the maximum theoretical speedup

```
task benchmark1 (depth):
    if(depth == 0):
        wait(DELAY_CYCLES)
    else:
        for(i in range(BRANCH_FACTOR)):
            wait(DELAY_CYCLES)
            spawn benchmark1(depth - 1)
```

Fig. 9: Benchmark 1 testing the work-stealing scheduler.

Evaluation environment. We use a SystemC-based cycle-accurate simulation environment for performance modelling. The generator is used for creating RTLs for various hardware configurations used by our benchmarks. Then, RTL is converted to SystemC modules using Verilator. We leverage *transaction-level modelling* (TLM) for memory transactions between the RTL and the SystemC modules. In particular, we use Xilinx-provided open-source AXI-to-TLM bridge modules for RTL-SystemC interfacing. Our simulation environment provides an approximate model of memory with configurable access latency and bandwidth. We choose the simulation parameters to mimic the HBM characteristics available on FPGA boards designed for data centres, such as Alveo U55C.

A. Benchmarks

Our benchmarks are based on the *knary* benchmark used to evaluate the original Cilk scheduler [5]. The benchmarks have two main controlling parameters, shown in Figures 9, 10, and 11, which are DEPTH and BRANCH_FACTOR. Those two parameters allow controlling the degree of parallelism of the workload by controlling the critical-path length and the amount of parallelism at each level of execution. Another parameter is SERIAL_TASKS, in Figure 11, which controls the number of dependent tasks within each DEPTH level where SERIAL_TASKS is less than or equal to the BRANCH_FACTOR. We set the parameters for the different benchmarks guided by the OpenCilk [7] analysis tool and select values corresponding to the targeted applications.

The benchmarks are designed to test different functionalities of our architecture. Benchmark 1, shown in Figure 9, is designed to evaluate the main scheduler component described in Section III-B. Benchmark 2, shown in Figure 10, is designed

```
task benchmark2_1 (depth):
    if(depth == 0):
        wait(DELAY_CYCLES/2)
        spawn benchmark2_2()
    else:
        for(i in range(BRANCH_FACTOR)):
            wait(DELAY_CYCLES/2)
            spawn benchmark2_2()
            spawn benchmark2_1(depth - 1)

task benchmark2_2():
    wait(DELAY_CYCLES/2)
```

Fig. 10: Benchmark 2 testing the system efficiency in accelerating applications composed of multiple types of tasks executed spatially by different sets of PEs.

```
task benchmark3 (depth):
    if(depth == 0):
        wait(DELAY_CYCLES)
    else
        for(i in range(iterator, BRANCH_FACTOR)):
            wait(DELAY_CYCLES)
            if(i < SERIAL_TASKS):
                spawn benchmark3(depth - 1)
                join()
            else:
                spawn benchmark3(depth - 1)
```

Fig. 11: Benchmark 3 testing continuation allocator and argument notifier. The join statement is used for ease of understanding; however, we implement it for our system with the spawn_next and send_argument primitives.

to test the efficiency of the interaction of two schedulers for two different tasks. Since our system provides the capability to divide an application among different types of PEs, we evaluate that performance does not degrade compared to having a single type of PEs that executes the whole task. This benchmark has a workload identical to Benchmark 1; however, the work is divided among two types of tasks and hardware PEs. Benchmark 3, shown in Figure 11, introduces a data dependency in the execution to evaluate the continuation allocator and the argument notifier. Adding a number of serial tasks on the execution path reduces the overall parallelism of the application; however, we ensure that the benchmark has a degree of parallelism an order of magnitude more than the compute units. We note that the spawn_next write buffer that accepts the successor task immediately is not yet part of the RTL output by our generator; for the results, we simulate its functionality as part of the PE.

B. Efficiency of the Scheduler

Efficiency of OpenCilk [7]. We implement the benchmarks in OpenCilk as a software baseline. For OpenCilk, the delay cycles correspond to a number of empty loops. We vary the delay from 16 loops to 1024 loops to simulate different workloads, and we report the efficiencies in Figure 12(a). The results show that for small tasks the scheduler overhead dominates the execution. This is not particularly surprising and explains some program refactoring for coarsening the tasks used in practical applications [11]. Even with big tasks the efficiency of execution for all the benchmarks converges to slightly less than 90%, probably because the scheduler logic in software subtracts a nonnegligible fraction of CPU compute power from tasks execution.

Efficiency of HardCilk. For our system, the delay is specified in clock cycles. We configure the system with 28 PEs to correspond to the 28 cores available for OpenCilk. We note that, even with the difference between the task implementation of the waiting (the work cycles) between software and hardware execution elements, the best-case parallelism in the application is constant because it depends on the ratio between the total work and the critical-path work. Therefore, the efficiency of OpenCilk could be compared qualitatively to that of HardCilk. We vary the task cycles from 8 cycles to 256 cycles to simulate different workload sizes. Figure 12(b) shows that HardCilk could achieve near-perfect efficiency when the PEs execute tasks lasting 32 cycles or more. Unlike OpenCilk, the scheduler is a separate circuit component and does not subtract compute cycles from the PEs: 100% efficiency is achievable. Tasks that execute for small number of cycles (8–16) take a stronger hit for Benchmarks 2 and 3 than Benchmark 1. Benchmark 2 indicates that using multiple types of PEs instead of monolithic PE is efficient for tasks as short as 32 cycles. Benchmark 3 has the lowest parallelism of the three, but HardCilk efficiency degradation with smaller tasks is still significantly better than that of OpenCilk.

C. Scalability of the Scheduler

Figure 13 compares the scalability of OpenCilk and HardCilk. Experiments run tasks that simulate work execution either in terms of empty loop iterations for OpenCilk or cycles for HardCilk. Simultaneous multithreading is turned off when the thread count is smaller than the number of CPU cores–28 in our case. OpenCilk shows almost linear scalability for up to 14 threads, after that point the task management overhead degrades scalability. With SMT turned on and using 56 execution threads, they compete for the resources of the same cores, further hampering the performance. In contrast, HardCilk shows linear scalability for up to 128 PEs. Only at 256 PEs, the scalability is sublinear and we attribute this to the exacerbated latency of the ring networks.

D. The Effect of System Servers Number on Efficiency

Figure 14(a) shows that HardCilk is capable of masking the effect of memory latency on task management by exploiting the available bandwidth with more scheduler servers. The experiment is executed using Benchmark 1, a task delay of 8 cycles, and 28 PEs. Figure 14(b) shows that HardCilk can significantly enhance the efficiency of the system by distributing atomic join counters access on a larger number of argument notifier servers. For this experiment, we use Benchmark 3, a task delay of 64 cycles, 28 PEs, and a memory latency of 35 cycles. With a faster memory, this graph will saturate faster, as in Figure 14(a). The results show that HardCilk can saturate

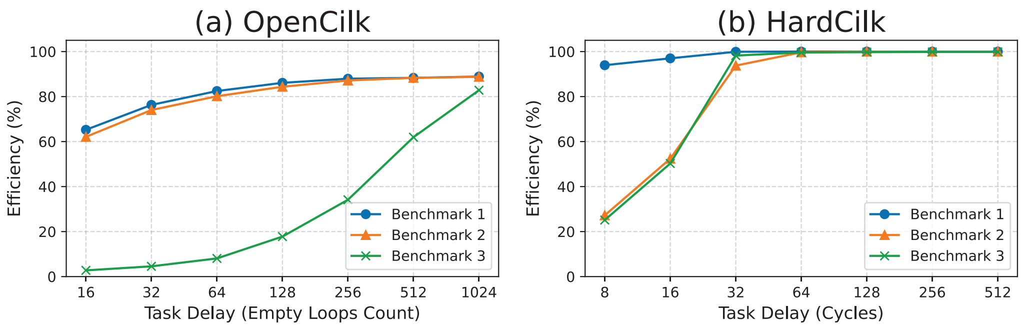

Fig. 12: Efficiencies of (a) OpenCilk [7] using 28 threads on a 28-core dual CPU system and (b) HardCilk with 28 PEs. For all benchmarks, OpenCilk efficiency saturates near 90% with the biggest task size while HardCilk already saturates close to 100% for tasks as short as 64 clock cycles. OpenCilk struggles more with Benchmark 3 as it has the lowest parallelism, while our system achieves better efficiencies with relatively smaller tasks.

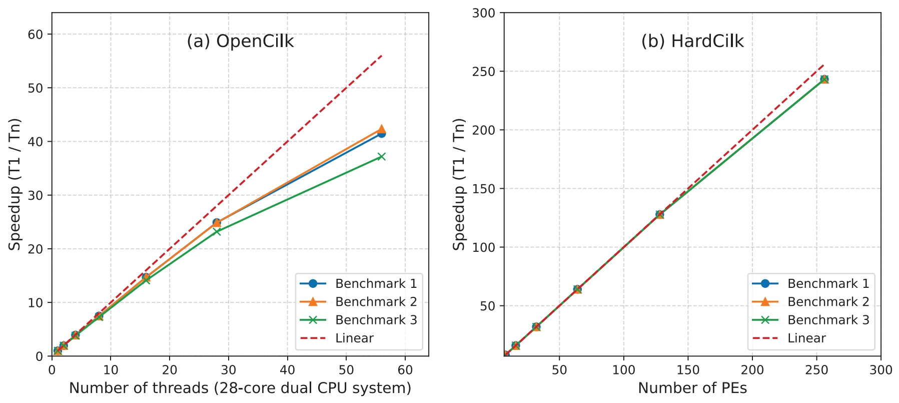

Fig. 13: Scalabilities of (a) OpenCilk with tasks of 1024 empty loop iterations and (b) HardCilk with tasks of 256 cycles. OpenCilk scales linearly up to 14 threads, after which task management overhead degrades scalability. The stability further degrades when exceeding the number of physical cores. In contrast, HardCilk scales linearly for up to 128 PEs.

the system and achieve around 98% efficiency with 8 argument notifier servers in this configuration.

E. Decoupled Access/Execute

Decoupled address/execute (DAE) is a classic technique to reduce the impact of memory-related stalls on program performance that has seen renewed interest in the design of hardware accelerators [12], [13], [14]. In a naïve program, there is a single type of unit that issues both memory accesses and performs computation. In case of a memory-related stall, the entire program stalls without doing any useful work. DAE addresses such stalls by introducing separate access and execute units: The access unit issues memory accesses, and the execute unit performs the computation. In case of a memory-related stall, only the access unit stops and the execution unit keeps doing useful work. A compiler generates two instruction streams for a given program, respectively for access and execute units; access and execute units communicate over a queue. We make the observation that HardCilk contains all the architectural elements required to implement DAE: the queue can be naturally implemented by our task queues and networks, and we support different types of PEs.

To demonstrate DAE on HardCilk and show how TLP is key to tolerate potential high latencies of memory hierarchies, we designed two experiments that traverse nodes in a linked data structure. The first experiment consists of a single type of PE that blocks on memory accesses while the second experiment has a single access PE and varied number of execute PEs. The task of an execute PE is a node: it performs a constant amount of computation (128 ns) and issues other execute tasks for linked nodes. In the blocking case, the execute PE blocks while the list of adjacent nodes is fetched from memory; in the DAE case, the access PE fetches them. The access PE issues nonblocking memory requests to take advantage of *memory-level parallelism* (MLP). We model the memory using memory delay (which we vary) and the maximum number of outstanding requests (i.e., its bandwidth, which we fix to 32).

The triangle-series in Figure 15 shows the execution time for the blocking case. The execution time shows a great sensitivity to the memory latency. This is because the MLP is limited by

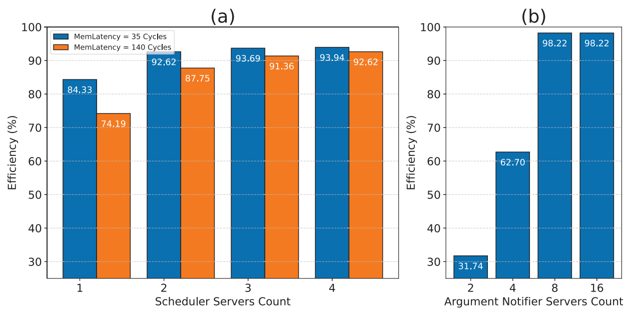

Fig. 14: System sensitivity to memory latency varying (a) the number of scheduler servers (Benchmark 1, task delay = 8 cycles, PE count = 28) and (b) the number of argument notifier servers (Benchmark 3, task delay = 64 cycles, PE count = 28, memLatency = 35 cycles). Experiments show that the system design can mitigate memory latency problems by exploiting bandwidth.

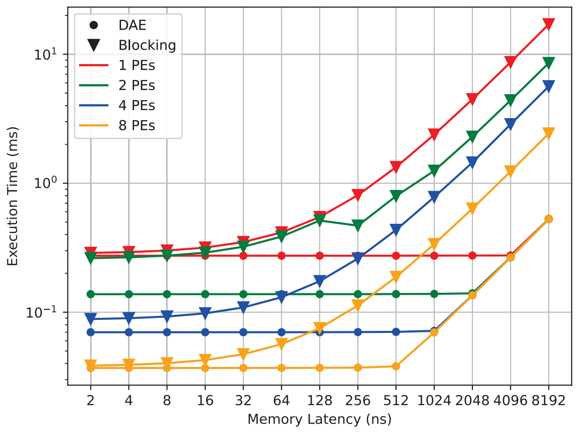

Fig. 15: Comparison of execution times for decoupled access/execute and blocking PE cases. For both, we fix the computation time to 128 ns and model a memory supporting 32 outstanding requests. We observe that while the execution time of the blocking PE exhibits a latency sensitivity, DAE execution time is latency-insensitive as long as the memory bandwidth is greater than the compute bandwidth.

the number of available execute PEs, making it impossible to take advantage of the full memory bandwidth. The point-series in Figure 15 shows the execution time for DAE, which is now insensitive to memory access latency as long as the memory bandwidth is sufficient. The memory bandwidth gets saturated because the access PE can exploit MLP to a greater extent.

The access PE can be implemented in an application-independent fashion. That is, our framework can provide a single access PE to be used by a large breadth of different applications. If a user were to identify critical long-latency loads, it would be easy for a compiler to split a task and achieve this insensitivity with an automatic code transformation.

TABLE III: Resource utilization. The first section lists the costs of individual components: C = client, S = server, Alloc = closure allocator, Sch = scheduler, Arg = argument notifier, q = task queue, and buf = buffer. The second section shows the total HardCilk hardware cost for different numbers of PEs running Benchmark 2.

	LUT	**Util.**	**FF**	**Util.**	**BRAM**	**Util.**
Avail.	**1,303,680**	**100.00%**	**2,607,360**	**100.00%**	**2,016**	**100.00%**
AllocC. + buf.	52	0.00%	147	0.01%	0	0.00%
AllocS.	793	0.06%	1,243	0.05%	0	0.00%
SchC. + q.	985	0.08%	969	0.04%	8	0.40%
SchS.	1,940	0.15%	556	0.02%	0	0.00%
ArgC. + buf.	112	0.01%	94	0.00%	0	0.00%
ArgS.	685	0.05%	801	0.03%	0	0.00%
8 PEs	13,283	1.02%	13,076	0.50%	64	3.17%
16 PEs	22,842	1.75%	21,785	0.84%	128	6.35%
32 PEs	42,079	3.23%	39,230	1.50%	256	12.70%
64 PEs	79,792	6.12%	74,027	2.84%	512	25.40%
128 PEs	155,172	11.90%	143,707	5.51%	1,024	50.79%

F. Synthesis Results

To evaluate the resource utilization of HardCilk, we synthesized the system in Benchmark 2 for AMD FPGAs. We report our synthesis results in Table III for the FPGA with part number `xcu280-fsvh2892-2L-e`, which is used by Alveo U280 data centre FPGA board, using Vivado 2022.2. Our design works at a clock frequency of 250 MHz.

For most resource types, utilization is a small fraction of those available. Figure 16 shows that the resource utilization increases linearly with the increasing number of PEs. However, BRAM utilization, compared to other resource types, is particularly high. The synthesized design contains task queues of 32 items for 256-bit tasks and task queues are the only type of resource that use BRAMs. Synthesis reports indicate that Vivado uses 8 BRAM primitives with a width of 32 bits and a depth of 1024 elements to implement them. This is not ideal: only a small fraction ($32/1024 = 3.125\%$) of the BRAM resources are actually utilized. However, our design relies on double-ended queues, for which BRAMs are the most suitable memory resource on FPGAs. We reckon that a more optimized implementation might access a single BRAM primitive in several cycles of a faster clock to share the resource across queue and improve the efficiency.

V. Related Work

A fairly small number of papers already explored the use of TLP on FPGAs. One of the first efforts implemented k-mean clustering with OpenCL atomics [15]. The paper performed load balancing via work-stealing managed on the host side, but host-side management of scheduling led to

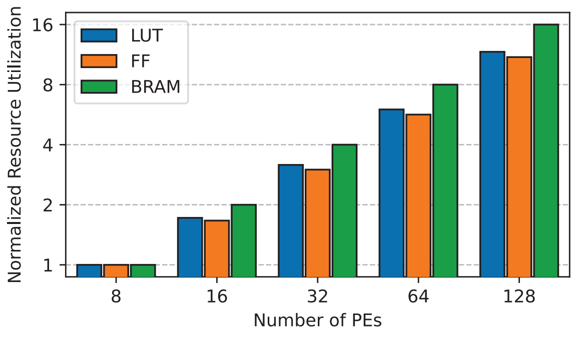

Fig. 16: Resource utilization relative to 8 PEs. As the number of PEs increase, resource utilization increases linearly.

a high performance overhead. TaPaSCo [16] used a similar software-side management of tasks but provided a tool flow to produce accelerators integrated into usable systems-on-chip. The accelerators were produced based on an HLS description of the accelerator core. Cascabel [10] tried to overcome the need for multiple round-trips between the host and the accelerators for task management. For this, it added on-chip task scheduling and launching capabilities to TaPaSCo. Nonetheless, Cascabel used a BRAM-based stack limiting the recursion depth, had generally limited flexibility, and blocked PEs for the duration of subtask recursive execution; the versatility was therefore unmatched to the expectations of software programmers. TAPAS [17] used pipelined dataflow execution units with dynamic task scheduling to execution tiles. Further, it supported recursive parallelism. Yet, the paper has no discussion of load balancing and load imbalance is visible in the results. TAPA [18] is similar to TAPAS, but uses a static mapping of tasks to execution units. The paper focuses more on reducing HLS code size and accelerating software simulation and hardware generation.

ParallelXL [3] is probably the piece of prior work that is closest to our goals. Like HardCilk, ParallelXL provides on-chip dynamic load balancing and dynamic task spawning. ParallelXL developed full hardware support for the execution model introduced in Cilk [5] and used HLS to generate accelerators from C++ descriptions of the tasks, much as we do. Yet, as we anticipated in the introduction, we feel that ParallelXL did not exploit some key opportunities to specialize in hardware key pieces of functionality, such as adding our dedicated stealing and continuation support for each task, as we instead did. On the other hand, although implemented in hardware, most of their architecture strongly reminds of a classic multicore architecture, with PEs implementing all possible tasks and the interconnect network loaded with all messages. We believe that the authors left on the table a significant potential that a reconfigurable TLP system could profitably exploit. Unfortunately, their design is neither open-source nor available directly from the authors, so we could not compare quantitatively to it.

Commercial HLS tools also provide limited support for TLP: Vivado by Xilinx/AMD, for instance, makes it possible to simply instantiate a single hardware unit per task (a C/C++ function) and connects these units using hardware FIFOs. There is no out-of-the-box solution for load balancing in case of multiple hardware units and it is thus more of a support for streaming processing. Moreover, deadlocks are possible: FIFOs implemented purely using FPGA resources are naturally bounded [19]. Clearly, with what existed prior to HardCilk we were still far from being able to run state-of-the-art task parallel software on FPGAs. We hope that this work is a significant step in this direction.

VI. Conclusion

In this paper we argue that software-like TLP deserves to be a first-class citizen in HLS tools targeting FPGAs. We think TLP management is a perfect task for efficient spatial implementation and have shown that it is possible to conceive a few relatively simple components and some dedicated networks to support very efficiently arbitrary combinations of tasks. In doing that, we have ensured that any critical hardware resource is virtualized into main memory, when needed, to give the same resource-agnostic experience one gets in software TLP. We exploit the massive parallelism of TLP applications to ensure that PEs are constantly busy irrespective of the latency of critical components such as DRAM. The result is HardCilk: an open-source Chisel generator [4] that creates the complete task management system for Cilk-like TLP applications.

We have built a set of synthetic workloads that gave insights into the efficiency of our architecture. In particular, we have shown that even our simple work-stealing and continuation components achieve an almost perfect efficiency for a broad set of execution parameters; all HardCilk needs is to scale up some critical resources depending on the application characteristics and the level of parallelism in the hardware. While we have modelled memory accesses from our system, we have deliberately excluded user memory accesses from most of our synthetic benchmarks, because the construction of an efficient memory system for TLP-like applications is a totally orthogonal problem. Yet, we have also shown that our efficient fine-grain TLP is perfectly apt to implement tasks so as to mask the memory system latency making memory bandwidth the accelerator bottleneck rather than latency; this can be achieved through simple code refactoring that most likely a compiler could automate.

Finally, we note that HardCilk is not yet integrated into a TLP compiler or library but that its integration seems perfectly feasible: Already, most parameters input to HardCilk are directly obtainable by static analysis of the TLP code whilst PEs can be completely designed with commercial HLS tools. Other sizing parameters that HardCilk needs to generate the full system may require some design-space exploration, but our experiments show that their effect is clear and predictable, boding well for simple heuristics. Ultimately, we aim to develop a comprehensive Cilk-like compiler for FPGAs, and we believe that HardCilk is its cornerstone.

References

[1] J. Blanuša, R. Stoica, P. Ienne, and K. Atasu, "Manycore clique enumeration with fast set intersections," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 2676–2690, jul 2020. [Online]. Available: https://doi.org/10.14778/3407790.3407853

[2] J. Blanusa, P. Ienne, and K. Atasu, "Scalable fine-grained parallel cycle enumeration algorithms," in *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 247–258. [Online]. Available: https://doi.org/10.1145/3490148.3538585

[3] T. Chen, S. Srinath, C. Batten, and G. E. Suh, "An Architectural Framework for Accelerating Dynamic Parallel Algorithms on Reconfigurable Hardware," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2018, pp. 55–67.

[4] M. Shahawy, C. Sönmez, C. Belentepe, and P. Ienne, "Hardcilk: Cilk-like task parallelism for fpgas," Apr. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.10971564

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, no. 8, p. 207–216, aug 1995. [Online]. Available: https://doi.org/10.1145/209937.209958

[6] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 212–223. [Online]. Available: https://doi.org/10.1145/277650.277725

[7] T. B. Schardl and I.-T. A. Lee, "Opencilk: A modular and extensible software infrastructure for fast task-parallel code," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 189–203. [Online]. Available: https://doi.org/10.1145/3572848.3577509

[8] C. F. Joerg, "The Cilk system for parallel multithreaded computing," Ph.D. Thesis, Massachussetts Institute of Technology, Boston, Mass., Jan. 1996.

[9] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[10] C. Heinz and A. Koch, "On-Chip and Distributed Dynamic Parallelism for Task-based Hardware Accelerators," *Journal of Signal Processing Systems*, vol. 94, no. 9, pp. 883–893, Sep. 2022. [Online]. Available: https://doi.org/10.1007/s11265-022-01759-2

[11] J. Blanuša, R. Stoica, P. Ienne, and K. Atasu, "Manycore clique enumeration with fast set intersections," *Proceedings of the VLDB Endowment*, vol. 13, no. 11, pp. 2676–90, Jul. 2020.

[12] J. E. Smith, "Decoupled access/execute computer architectures," *ACM Transactions on Computing Systems (TCS)*, vol. 2, no. 4, pp. 289–308, Nov. 1984.

[13] T. Chen and G. E. Suh, "Efficient data supply for hardware accelerators with prefetching and access/execute decoupling," in *Proceedings of the 49th Annual International Symposium on Microarchitecture*, Taipei, Taiwan, Oct. 2016, pp. 1–12.

[14] T. J. Ham, J. L. Aragón, and M. Martonosi, "Decoupling data supply from computation for latency-tolerant communication in heterogeneous architectures," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 16:1–16:27, Jun. 2017.

[15] N. Ramanathan, J. Wickerson, F. Winterstein, and G. A. Constantinides, "A Case for Work-stealing on FPGAs with OpenCL Atomics," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Monterey California USA: ACM, Feb. 2016, pp. 48–53. [Online]. Available: https://dl.acm.org/doi/10.1145/2847263.2847343

[16] C. Heinz, J. Hofmann, J. Korinth, L. Sommer, L. Weber, and A. Koch, "The TaPaSCo Open-Source Toolflow: for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems," *Journal of Signal Processing Systems*, vol. 93, no. 5, pp. 545–563, May 2021. [Online]. Available: https://link.springer.com/10.1007/s11265-021-01640-8

[17] S. Margerm, A. Sharifian, A. Guha, A. Shriraman, and G. Pokam, "TAPAS: Generating Parallel Accelerators from Parallel Programs," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2018, pp. 245–257.

[18] Y. Chi, L. Guo, J. Lau, Y. Choi, J. Wang, and J. Cong, "Extending high-level synthesis for task-parallel programs," in *29th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2021, Orlando, FL, USA, May 9-12, 2021*. IEEE, 2021, pp. 204–213. [Online]. Available: https://doi.org/10.1109/FCCM51124.2021.00032

[19] Xilinx, "Abstract parallel programming model for hls," https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Abstract-Parallel-Programming-Model-for-HLS, accessed on: March 15, 2023.